UNIVERSIDAD AUTONÓMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

Degree in Computer Engineering

TRABAJO FIN DE GRADO

BACHELOR'S THESIS

Desarrollo de un videojuego sobre un motor JavaScript

Development of a videogame using a JavaScript engine

Silvia Barbero Rodríguez

Tutor: Carlos Aguirre Maeso

Mayo de 2017

May 2017

Development of a videogame using a JavaScript engine

Desarrollo de un videojuego sobre un motor JavaScript

Silvia Barbero Rodríguez

TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior

Universidad Autónoma de Madrid

May 2017

Resumen

Este Trabajo Fin de Grado explica los primeros pasos del desarrollo del juego Star Quest: Treasure Time, un metroidvania multijugador protagonizado por dos personajes con varias habilidades diferentes, que necesitarán combinar esas habilidades para avanzar por muchos niveles. El juego está hecho con el motor Clockwork.js , un motor en JavaScript creado y presentado como Trabajo Fin de Grado por un compañero, y será publicado en la tienda de Windows y, si es posible, en Xbox One. Este proyecto ha sido creado para ser lo suficientemente bueno para presentar a varias competiciones de videojuegos, y para usarse como demostración de la capacidad del motor, siendo un juego lo suficientemente complejo para mostrar todo su potencial. Para ello, será una experiencia divertida con muchas habilidades que los jugadores tendrán que usar para seguir adelante, una obra de arte con grandes mapas y personajes animados tradicionalmente, y un innovador proyecto que incluirá las características propias de cualquier metroidvania y añadirá nuevas como el multijugador cooperativo o la iluminación dinámica.

Este documento describirá el diseño del juego: la apariencia de los personajes, sus habilidades y personalidad, la historia del juego y la motivación de los personajes, y los módulos en los que se divide el código con explicaciones de la función de cada uno. Después de esto, se discutirá brevemente el estado del arte de juegos parecidos: metroidvanias, y juegos de Universal Windows Platform. El desarrollo de las características más importantes será explicado a continuación, seguido del proceso de integración y pruebas de las mismas, y una sección sobre las conclusiones extraídas del desarrollo y el futuro del juego.

Abstract

This Bachelor's Thesis explains the early development of the game *Star Quest: Treasure Time*, a multiplayer metroidvania starring two main characters with various different abilities, who will need to combine their strengths to advance through many levels. The game is made with the JavaScript engine Clockwork.js, developed and presented as Bachelor's Thesis by a colleague, and will be published in the Windows Store and hopefully Xbox One. It is meant to be a project worth of presenting to various game competitions as well as a demo for the engine used which is complex enough to showcase its potential. For that, it will be a fun experience with many abilities the players will have to use in order to advance, a beautiful piece of art with big maps and traditionally animated characters, and an innovative project that includes the features every metroidvania game has and adds new ones like the cooperative multiplayer or dynamic lighting.

The document will describe the game's design: the character's looks, abilities and personalities, the story of the game and motives of the characters, and the modules the code is divided into with explanations of their function. After this, there will be a brief discussion on the state of the art of games like this one: metroidvanias, and Universal Windows Platform games. The development of the most important features of the project will then be explained, followed by the process of integration and testing of those features, and a section on the conclusions extracted from the development and the future of the game.

Palabras clave

Juego, videojuego, desarrollo de videojuegos, plataformas, metroidvania, JavaScript, motor Clockwork, UWP, Universal Windows Platform, Xbox.

Keywords

Game, videogame, game development, gamedev, platformer, metroidvania, JavaScript, Clockwork engine, UWP, Universal Windows Platform, Xbox.

Acknowledgments

I would like to thank my colleague Arcadio García Salvadores for the development of his game engine Clockwork.js; without it none of this would have been possible.

I would also like to thank my tutor Carlos for reviewing this document.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

This Bachelor Thesis explains the early development of the game *Star Quest: Treasure Time*, a platformer game with two main characters which will be controllable by two different players who will need to cooperate in order to advance through the levels, by either using each of the character's abilities where needed, or combining them. To be more specific, the game could be classified as a metroidvania, a subgenre of action-adventure games which includes features reminiscent to the games *Metroid* and *Castlevania*, having a collection of unlockable abilities, and being the different levels in the world interconnected for the player to roam freely, sometimes needing some of those unlockable skills to enter or advance through them.

It is being built with the engine *Clockwork.js*, "an open platform for developing HTML5 games based on modular components" [1], created by my fellow colleague Arcadio García Salvadores and presented as his Bachelor Thesis this same year (2017), and its default rendering library *Spritesheet.js*[2]. This engine is based on web technologies using JavaScript for all the code, and JSON or XML for the information on the levels and spritesheets (in my case it will be XML). This engine provides only the basic functionality needed to create games such as a setup event, executed when game objects are created, a loop event called each frame, modules for including keyboard, touch and gamepad controls, …, but does not provide any physics, prefabs, or GUI.

MOTIVATION

The motivation behind this project was to create a game to compete in various game development competitions. The main goal was for it to be an entry for *Imagine Cup*[3], Microsoft's most important tech competition for students. However, this year the games category was taken away and the three original categories (games, innovation, and world citizenship) were merged into one, leaving games with almost no possibility to win, being mere entertainment products, having to face projects that solve real-life problems. Given this situation, I decided to prepare the game for the *Game Development World Championship*[4] (GDWC), an indie game development contest. Apart from this, I will also prepare it for the Intel Level Up Contest[5] if it is celebrated next year, in which judging has in mind aspects like art and character design in which I didn't focus as much for this Bachelor's Thesis, as it is my computer engineering skills what I must showcase.

For competing, I wanted to create a metroidvania game with cooperative multiplayer characteristics that could be played on both PC and Xbox thanks to the Universal Windows Platform. As games build with Clockwork are made entirely with web technologies, the engine provides bridges for games to be exported to the web, or as UWP applications that can be used on all devices running Windows 10 (Computers, Xbox, Phones with Windows 10 Mobile, HoloLens or other devices with Windows Holographic, Raspberry Pis, … and new devices to come) if the developer desires so.

Another reason for this game to be developed is to serve as an example of what Clockwork can do beyond the easy demos created to show its features, a more complex project that could push the limits of the engine to show its potential and compete with other well-known engines, as well as to unveil any bugs or deficiencies to be fixed in the engine during the development.

OBJECTIVES

As a project to be presented to game competitions, its objectives were:

- To create a cooperative game which is fun, easy to play while still challenging, keeps players interested and feels good to play.
- To create a game that is appealing, with many abilities, big, beautiful backgrounds and charismatic characters with smooth animations.
- To create a game that is innovative, by making it a well-known genre with new features.

ORGANIZATION OF THE DOCUMENT

After this introduction, the document begins with a brief section on the state of the art of metroidvania games and Universal Windows Platform games.

After it, the design of the game is discussed, going from the main character design and story to the structural design.

This is followed by the largest section in the document: the one describing the development of the project. It explains how the game started and how the most significant features of it were implemented.

Finally, the description of the integration, tests, and result follows, and the conclusion and future of the project end the document. After this, references, glossary and annexes follow.

BRIEF INTRODUCTION TO CLOCKWORK.JS

In order to fully understand this document a basic knowledge of the engine used is needed. Therefore, this section provides an introduction to Clockwork.

Every Clockwork project starts with a manifest where the information to run the game is. It registers the components, levels, spritesheets and dependencies to be used, and it is where the screen resolution, frames per second, project name, and other features are chosen.

Clockwork Runtime interacts with the operating system or the web browser, reading the manifest to set everything up. Then Clockwork Core loads the first level in levels.xml, the file where all the information on levels is, and spawns all the components specified in that level. Components are most of the code of your specific game, they implement the game logic, interact with native APIs, and process the input. They can communicate between them through message passing thanks to Clockwork Core, which also sends the relevant changes in their inner states to the rendering pipeline. Finally, this rendering pipeline is a stack of libraries that processes the received data, and renders it using the specified spritesheets.

There are a few events that start with #, which can be implemented by components and have special behaviors. *#setup* is an event that is automatically called when the object is created, *#loop* is executed each frame, and *#collide* is triggered when something touches the object. There are also variables starting with $ that can be read by Spritesheet and used in spritesheets.xml, where all the information about the spritesheets is. The default ones are *$x*, *$y, and $z*, which represent the position of the component, and *$state*, which is the animation state the component is in, but the user can define more.

NOTES TO HAVE IN MIND

- Over the document, I will show pieces of code extracted from the game. If you've seen Clockwork's documentation you might find a different syntax from the one shown in some examples. This is due to the fact some parts of the game were coded in a previous version of Clockwork, before the game was ported to the new one.

- You will also see art in several stages. Characters went through digital mockup and traditional sketch phases before reaching their final look. Background art went through a few iterations too.

- A demo can be played at http://treasuretime.azurewebsites.net/ . The controls can be found in annex A at the end of this document.

STATE OF THE ART

METROIDVANIA

There are quite a few metroidvania games in the market, a simple Steam search will reveal some of the most popular ones[6]. Some examples of metroidvania games are:

SHANTAE: HALF-GENIE HERO

Released on the 20th of December of 2016, this game is about a half-genie heroine with the ability to transform in various creatures by dancing. The dances are unlocked throughout the game, at the end of story levels or in secret rooms, and are needed to get through certain obstacles which can either be in the next worlds or in previous ones, which need to be revisited to find hidden objects. It is currently owned by more than 50,000[7] users on Steam but is also available on PlayStation 4, PlayStation Vita, Nintendo WiiU, Xbox One, and soon on Nintendo Switch. Due to its Kickstarter campaign, it built a lot of hype and its launch was a great success.

OWLBOY

A game about an owl boy with the ability to fly and hold other characters. It was released on the 1st of November of 2016 and features a gigantic world and many gunners who are met throughout the story and can be summoned by the protagonist to be held and carried by him, providing different abilities. It is currently owned by more than 70,000[8] users on Steam but it is also available on the Humble Store, Good Old Games, and Indiebox. Its release was also very successful due to the hype built through its 9-year development.

ORI AND THE BLIND FOREST

This beautiful game was released on the 11th of March of 2015 and narrates the story of an orphan who must save the forest by using different abilities acquired throughout the story. This game is especially memorable for its stunning artwork. It counts with more than 900,000[9] owners on Steam but it's also available for Xbox One.

There are many others, of course, but to my eyes, these are the latest, most charismatic ones. They are very successful indie titles, making them projects to learn from when developing a similar one. Like all of these, *Star Quest: Treasure Time* will feature many abilities to be unlocked in order to advance or find secrets, together with a well-crafted story and a memorable art style.

MULTIPLAYER METROIDVANIA

However, there aren't many cooperative multiplayer metroidvania games, and the ones available don't really require the cooperation. That is where Treasure Time is innovative, in that the whole gameplay is focused on the two players needing each other to advance, and having truly unique abilities that make the characters be really different, feeling more alive.

UNIVERSAL WINDOWS PLATFORM AND WEB

Another important feature of *Star Quest: Treasure Time* is that it will be made using the Clockwork Platform, and be the first important game created with it. This will give it exposure because it will be the main game to showcase the engine's potential and the first complex one available in the platform.

Clockwork will make the game easily available for the Universal Windows Platform, which allows it to be played in devices running Windows 10 and Xbox One, being acquired from the Windows Store and enabling players to use it in all devices with just one purchase with the Xbox Play Anywhere program[10]. There are already many games in this market, from traditional mobile games to console titles.

Clockwork can also export games to webs, to be played with any browser. There are many games that can be played in web browsers but none with the level of complexity *Treasure Time* will have. This makes this project be innovative in that aspect.

DESIGN

THE CHARACTERS

Despite being the code the most important part for a Bachelor's Thesis for a Computer Engineering student, I need to start explaining the characters, as they were the theme over which the rest of the game was born, and I will mention them over the rest of the document quite a few times.

ESSIE

A treasure hunter with an unhealthy thirst for power. She might look cute and harmless but she's strong-minded and fearless. At the start of the game, Essie has only two abilities: crouching to get through small holes, and opening locks on chests and similar objects. However, she has a deep understanding of magical artifacts and is eager to try and master any she finds.

During the story, several of those artifacts, known as "Fallen Stars" in-game, will be found by the player, and Essie will be able to use them to unlock new abilities. Some examples would be the "Mermaid Pareo", that allows her to dive in water, the "Drill Hands" she will use to break cracked walls, the "Bat Ears" to "see" in the darkness, or the "Torchshade" to cast darkness.

Figure 1: Essie

KEYSTROKE

A writer cursed with wings that allow him to fly but make him turn to stone under daylight. He is usually very happy and loves puns. Flying will be needed by players to get to otherwise unreachable places but the sunlight curse will keep him from accessing other areas.

As the game advances, he will be able to learn more things he can do with his wings like flapping them strongly to create a gust, and he will find a few artifacts he can use too like the "Ink Sword", that lets him cut through plants blocking the way, and attack causing different effects depending on the ink it's using.

Figure 2: Keystroke

TOGETHER

Essie and Keystroke will need to combine their abilities to get through the levels. For instance, there might be a hole in the ceiling of a cave through which sunlight comes in and blocks the way for Keystroke. But right ahead there might be a huge rock Essie can't get through which blocks a locked door. In this case, Essie will have to pick up Key, carry him through the light and then let Key pick her up to fly her over the rock. Then Essie would open the locked door so they can both go through it.

STORY

In a world where magical creatures outnumber humans, Essie finds herself to be too weak to survive and seeks the *Fallen Stars*, powerful artifacts created from wishes upon stars fallen from the skies. These objects will grant her the strength she believes she needs to fight her fears. Very soon in her adventure, she meets Keystroke, who is in search of an artifact that can help him with his stone curse, perhaps one of the *Fallen Stars* that is known to grant control over rocks could help.

Although Essie denies any help from the kind Keystroke when they first meet, they soon find themselves to work a lot better as a team. This way, Essie and Keystroke start their adventure in search of *Fallen Stars* together.

GAME DESIGN

The game code is divided in different files according to what each component belongs to. As the game is right now, these files are:

- *characters.js*: All the components related to the characters, both the controllable ones and the enemies. These components are the following:
    - The most important component here is *mob*, which takes care of updating the position and sprite state of all the characters or moving objects.
    - The *character* component inherits *mob* and adds a few variables and events that allow characters to attack, take damage, and die.
    - A few other components that implement certain abilities and can be inherited in the specific characters, like having a health bar, jumping, crouching, attacking in different ways, holding, being held, or being playable.
    - The specific characters like Essie and Keystroke, that overwrite the default values of speed, width, height, etc, and add events that only they can execute, like the abilities given by the *Fallen Stars*.
- *interface.js*: The components related to the interface. These include things like the health bar, the money counter, the text for taken damage, the object selector, or the camera.
- *world.js*: The components related to the world, such as walls, particle systems, and background.
    - A *hitbox* component is the main protagonist here. It is a component with a box collider whose width and height are set in an event *setCollision* with the variables specified in the components which inherit it. It also provides an event *generateTexture*, which sets the texture of the object as it will be seen later in this document, and a *setWalls* event that sets walls through which light can't go through in all the sides of the hitbox, which will also be explained later.
    - A *solid* component which inherits *hitbox* and makes it a solid floor or wall.
    - *water* and *deepWater* components that inherit *hitbox* and have different behaviours to be shallow water characters can walk through and deep water characters can swim in, respectively.

    - *crackedWalls* and *plantWalls* are included in here too. They inherit *hitbox* and are solid until touched with a drill or sword attack respectively, then they are destroyed. *chest*s are another kind of component inheriting hitbox that acts as solid until touched by Essie, and then its content appears.

    - The *particles* are another important component, inherited by the many kinds of particle effects in the game, like stars, dust, water splashes or money.

- *light.js*: Here lies the *lighting* component for controlling all the dynamic lighting as well as a *light* component that creates a light source.

- *commonFunctions.js*: This file has just one component which implements some common functions, which will be used in other components frequently, as engine variables.

- *gameSetup.js*: This file contains just one component too, which is in charge of setting some needed engine variables and preparing the game to be played before loading the first level.

- *network.js*: The *server* and *client* components in charge of the connection for the multiplayer are here.

- *mirroringServer.js*: A rendering library which sends all the rendering data to the mirroring client before passing it to Spritesheet.js.

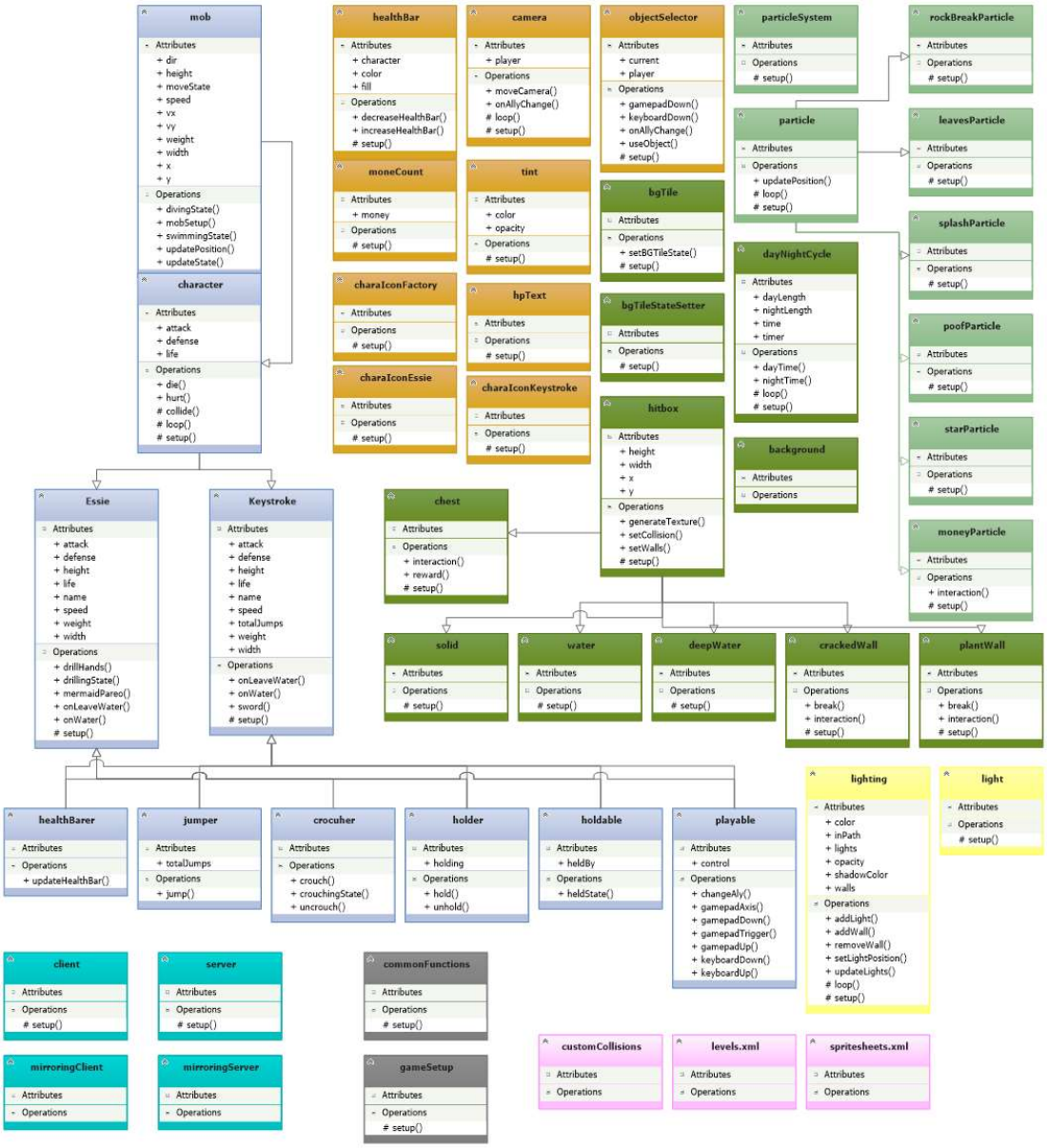
- *mirroringClient.js*: Another rendering library which receives the rendering data from the mirroring server and sends it to Spritesheet.js.

- *customCollisions.js*: A file containing the custom collision types created for this game, which are not included in components provided by Clockwork.

- *levels.xml*: All the levels of the game. In the future, they will be separated into more files to organize them by the parts of the game they are in but there aren't enough just yet.

- *spritesheets.xml*: The information of all the spritesheets of the game. This will eventually be divided into a few other files to separate spritesheets of characters, interface, world, …

The project's class diagram would be similar to the following but some components have been omitted and the ones shown have only the most important variables and events.


mob
Attributes
+ dir
+ height
+ moveState
+ speed
+ vx
+ vy
+ weight
+ width
+ x
+ y
Operations
+ divingState()
+ mobSetup()
+ swimmingState()
+ updatePosition()
+ updateState()
character
Attributes
+ attack
+ defense
+ life
Operations
+ die()
+ hurt()
# collide()
# loop()
# setup()
healthBar
Attributes
+ character
+ color
+ fill
Operations
+ decreaseHealthBar()
+ increaseHealthBar()
# setup()
moneCount
Attributes
+ money
Operations
setup()
charaIconFactory
Attributes
Operations
setup()
charaIconEssie
Attributes
Operations
setup()
camera
Attributes
+ player
Operations
+ moveCamera()
+ onAllyChange()
# loop()
# setup()
tint
Attributes
+ color
+ opacity
Operations
setup()
hpText
Attributes
Operations
setup()
charaIconKeystroke
Attributes
Operations
setup()
objectSelector
Attributes
+ current
+ player
Operations
+ gamepadDown()
+ keyboardDown()
+ onAllyChange()
+ useObject()
# setup()
bgTile
Attributes
Operations
+ setBGTileState()
# setup()
bgTileStateSetter
Attributes
Operations
setup()
hitbox
Attributes
+ height
+ width
+ x
+ y
Operations
+ generateTexture()
+ setCollision()
+ setWalls()
# setup()
particleSystem
Attributes
Operations
setup()
particle
Attributes
Operations
+ updatePosition()
# loop()
# setup()
dayNightCycle
Attributes
+ dayLength
+ nightLength
+ time
+ timer
Operations
+ dayTime()
+ nightTime()
# loop()
# setup()
background
Attributes
Operations
rockBreakParticle
Attributes
Operations
setup()
leavesParticle
Attributes
Operations
setup()
splashParticle
Attributes
Operations
setup()
poofParticle
Attributes
Operations
setup()
starParticle
Attributes
Operations
setup()
moneyParticle
Attributes
Operations
+ interaction()
# setup()
Essie
Attributes
+ attack
+ defense
+ height
+ life
+ name
+ speed
+ weight
+ width
Operations
+ drillHands()
+ drillingState()
+ mermaidPareo()
+ onLeaveWater()
+ onWater()
# setup()
Keystroke
Attributes
+ attack
+ defense
+ height
+ life
+ name
+ speed
+ totalJumps
+ weight
+ width
Operations
+ onLeaveWater()
+ onWater()
+ sword()
# setup()
chest
Attributes
Operations
+ interaction()
+ reward()
# setup()
solid
Attributes
Operations
setup()
water
Attributes
Operations
setup()
deepWater
Attributes
Operations
setup()
crackedWall
Attributes
Operations
+ break()
+ interaction()
# setup()
plantWall
Attributes
Operations
+ break()
+ interaction()
# setup()
healthBarer
Attributes
Operations
+ updateHealthBar()
jumper
Attributes
+ totalJumps
Operations
+ jump()
crocuher
Attributes
Operations
+ crouch()
+ crouchingState()
+ uncrouch()
holder
Attributes
+ holding
Operations
+ hold()
+ unhold()
holdable
Attributes
+ heldBy
Operations
+ heldState()
playable
Attributes
+ control
Operations
+ changeAlly()
+ gamepadAxis()
+ gamepadDown()
+ gamepadTrigger()
+ gamepadUp()
+ keyboardDown()
+ keyboardUp()
lighting
Attributes
+ color
+ inPath
+ lights
+ opacity
+ shadowColor
+ walls
Operations
+ addLight()
+ addWall()
+ removeWall()
+ setLightPosition()
+ updateLights()
# loop()
# setup()
light
Attributes
Operations
setup()
client
Attributes
Operations
setup()
server
Attributes
Operations
setup()
commonFunctions
Attributes
Operations
setup()
mirroringClient
Attributes
Operations
mirroringServer
Attributes
Operations
gameSetup
Attributes
Operations
setup()
customCollisions
Attributes
Operations
levels.xml
Attributes
Operations
spritesheets.xml
Attributes
Operations

DEVELOPMENT

FIRST STEPS

THE ORIGINAL IDEA

The idea for this project actually started in the first days of September when I decided to make this game in my free time and wasn't planned to be my Bachelor's Thesis project or to compete. It was just a fun experiment.

Three days later I found myself with a simple level with two characters able to run around, jump, fly, swim, hold each other and attack with a 3-part combo, uppercut, or slide attack. There were also keyboard and touch controls, some textures, a day-night cycle, some weather effects like falling snowflakes, a health bar, fall damage, floating numbers that showed the damage taken, and even a blood particle effect.

Figure 3: Essie's uppercut attack in a first prototype and screenshot of second prototype with some art, effects and new features.

There were a lot of features, but it was all chaos. Nothing had been planned beforehand; it was all the result of "what I felt like doing". Of course, once I decided to make it my Bachelor's Thesis project I had to make it more professional so it could be a real game.

The first thing I needed to do to transform this mess of a game into something I could really work with and expand in an organized and safe way was to redesign it and choose what features to keep and which to, sadly, forget about, and think how everything would be before rushing to type the code. This lead to a lot of refactoring to remake everything that I had in a much cleaner way to make it more efficient and modular.

The most important parts refactored were the ones involved with the character basic movement and jumping, holding, the day-night cycle, and the texture generation. Swimming was redesigned to make characters move only on the surface and rise to the surface if they were underwater instead of sinking and walking at the bottom like they did before; diving was made a separate ability. The overcomplicated attacks were removed to make attacking only possible with artifacts in their own special ways. Touch controls were removed for how complex they would be, and gamepad compatibility was implemented. The rest of the features were removed for the first prototype of the real game, which is meant to become a minimum viable product.

The next sections describe how the most important parts of the game were made. I've tried to order them by date but in most cases I worked on several Git branches at the same time, developing several features in parallel.

MOB

The first big step of this refactoring was the way the characters move. I made a Clockwork component named *mob*, from which all moving things would inherit its setup, *updatePosition*, and *updateState* events. All mobs have a set of variables that are given a default value in the setup: *width* and *height*, a *weight* and *speed* that can be overwritten in the specific mobs, a *jumping* variable to control if the character is jumping, and two variables named *vx* and *vy* to express the direction as well as a *dir* variable to indicate the direction of the sprite. To further explain:

- The variables *width* and *height* are the size in pixels of the character, the size I use for the collision box. These dimensions are not the ones of the sprite as I leave a considerable margin for the player to feel the collision "fair". That is to say, I prefer the image to be slightly over the walls, enemies, etc., than to infuriate players who are stopped or hit by them too soon.

- The *weight* is basically the speed with which the character will fall. For example, Keystroke has a lower value of it because he will fall slower than Essie does.

- The *speed*, as it can be easily guessed, is how fast the characters move.

- *jumping* is set to a specific value when a character jumps, and decreases by one each frame. Once it is 0, the character's jump force is over and it will proceed to fall. It also influences the speed at which the character goes up while jumping, being a slower movement each frame.

- *vx* and *vy* have values from -1 to 1 that are set when the character wants to move. For example, in the case of the main characters, when pressing the left arrow in the keyboard, *vx* will be set to -1, when pressing right, to 1. *vy* will most of the time be set to 1 (remember the HTML5 canvas has the origin point at the top left of the screen) to make the characters move down constantly unless there is something below, imitating gravity.

- *dir* is used to know in which direction to paint the sprite, being "L" and "R", for left and right respectively, all the possible values. I'll explain this a bit more later when explaining *updateState*.

Inside the *updatePosition* event, which be called inside the *#loop* event of its children to be executed each frame, I create a uniform rectilinear motion for the movement on the horizontal axis. With *x* being the character's position and *v* its speed, and *t* the time, the position of the character at a given moment follows the equation $x(t) = x + vt$. To do this, *vx* is multiplied times the speed to get the velocity of the character and so get the distance it will move and the direction in which it will do it. This will later be added to its current position.

The vertical axis, however, presents a rectilinear movement with an acceleration to mimic gravity. The speed of a jumping character can be compared with a parabola described by $y = x^2$, where the speed starts being negative and slows down until reaching zero, before starting to speed up again. This way, the character jumps with strength and gets slowed down as it reaches the end of the jump and then begins to fall faster. To mimic this behavior, *vy* is multiplied times the weight minus the jump strength and the *jumping* variable is decreased by one until it reaches 0.

For this piece of code and the ones to follow, have in mind that *this.getVar* is a Clockwork function that returns the value of the variable of the current object with the given name and *this.setVar* sets the value of the variable with the name passed as first argument to the second argument. These variables come from the first version of the engine and are now deprecated.

```
if (this.getVar("jumping") > 0) {
    this.setVar("jumping", this.getVar("jumping") - 1);
}
var vx = this.getVar("vx"); vy = this.getVar("vy");

vx *= this.getVar("speed");
vy *= this.getVar("weight") - this.getVar("jumping");
```

But this is only for the movement itself! Now the character needs to check if there are any walls blocking the way. To do so, I use Clockwork's *collisionQuery* function, which checks if there is any collision that hits the given one. In this case, I search for anything hitting a box collider the size of the character in the position in which it will next be. It is important to know the point from which the character *x* and *y* start to count is set in the middle of the lower side of it.

Figure 4: The point (0,0) of characters is set where the yellow circle is.

For checking if there are collisions in the *x* axis, the type of collision used to check for objects hitting the player is a box with its *x* starting at the player's *x* minus half of the width and plus the character's *vx*, its *y* starting at the player's *y* minus the height, and the width and height of the character.

For collisions in the *y* axis, the type of collision used is a box with its *x* starting at the player's *x* minus half of the width, its *y* starting at the player's *y* minus the height and plus the character's *vy*, and the width and height of the character.

The function returns an array of the objects that collide so the next step is filtering them to find those which are solid. If the array resulting from the filtering is empty, that means I can move. In the case of the *x* axis, I simply add *vx* to the character's *x*.

For the *y* axis, it is almost the same except that if the array isn't empty, it means the character has hit the ground (or ceiling depending on its current *vy*) so a *grounded* variable is set to true (or a *ceiling* variable which existed in one of the first prototypes but was removed because no uses where found for it but could be added any moment if it becomes useful). That variable is then used in other actions like jumping to only allow the character to do so if it is on the ground. There are also a few more lines for counting how many frames has the character been falling (not grounded) to control the fall damage, and to count the number of jumps done as some characters will be able to jump once or more in the air but the default behavior is to jump once and need to touch the ground before being able to jump again. Keystroke's flying, for example, is coded as an infinite number of jumps, and Essie usually can jump only once but when holding Key, she gets an extra jump she can execute in the air.

In *updateState* I take care of the sprite state. I check if I'm going left or right with *vx,* and I set *dir* accordingly, and then I set a sprite state depending on its moving state. If the character is grounded and its *vx* is different

from zero, the moving state is *Run*, if it's not, it is *Idle*. If the character is not grounded and it is jumping, the state is *Jump* but if the *jumping* variable is 0, the state is *Fall*. Then I simply append *dir* to that state to tell Spritesheet.js the state I want to use.

After this, I did the character component, which is basically a *#loop* event that calls both *updatePosition* and *updateState* under certain conditions defined by the state of the character (if it's petrified, hurt, etc,..), and takes care of the timers that indicate how long some states are applied (time hurt, time the attacking state lasts, …).

ALLY CHANGING

Despite the game being multiplayer, it would be nice for lonely users to play it by themselves, plus it would also be useful for me to test everything on my own. This way, one of the first features to be implemented was ally changing. In singleplayer mode, this feature is vital, as the game requires both characters to go through many levels, sometimes for their abilities and sometimes for staying at two places at a time, like buttons, for example. One approach would be to switch between characters having always just one on screen (like if they transformed into the other) but that would ruin most of the puzzles so the method used is to have both characters on screen and, by pressing a button or key, let the controls and camera change from one character to another.

All playable characters inherit a *playable* component which has exclusively the events called when the keyboard or gamepad is used and trigger the corresponding actions depending on the key or button pressed, so I simply gave Essie and Keystroke a *focus* variable which would only be *true* if the focus is in the character, that is to say, if that character is the one in control. Having that, I simply ignore everything if it's *false*.

To change I set the current character focus to *false* and the other one to *true*. But which one is the other one? When including the characters in levels.xml, I give them a variable called *nextAlly* which holds the name of the character it will switch to, and use this with Clockwork's *find* function to find the object with that name. The XML lines in the levels file look like:

```
<!--Charas-->
<object name="Essie" type="Essie" x="1500" y="600" vars='{"focus": true,
"nextAlly":"Keystroke"}'></object>

<object name="Keystroke" type="Key" x="1000" y="600" vars='{"focus": false,
"nextAlly":"Essie"}'></object>
```

The idea behind this is to be able to make the character changing as generic as possible by creating a circular linked list, as there could be someone else to control in the future apart from Essie and Key.

For example, in this case, Essie would have Keystroke as next ally, and Keystroke would have Essie but if there was a third character I could make Key have that one, and that one would have Essie, or in whichever order I wish. I can also make them unable to change character in this way by overwriting that variable with something else.

The changing event starts with a first check to see if there exists a *nextAlly* and if the *changeTimer* is 0. This timer is set in both characters to an arbitrary number of frames that have to pass before the player can change again. This is to avoid detecting the button or key press more than once while it's being pressed. Here I also set *vx* to 0 so that characters have to stop when changing, avoiding like this bugs like the character keeping its direction and speed and running away when it is not even being controlled. When all of this is done, *onAllyChange* is called globally (*this.engine.execute_event(<eventName>)* or *this.engine.do.<eventName>* in the new version, instead of *this.execute_event(<eventName>)* or *this.do.<eventName>*) so that any

component that has it, executes it. The event is called with a parameter (ally) that says what character it changed to.

The next step was to make the camera follow the character that is in control.

The camera is just a component that calls Spritesheet.js's *setCamera* function every frame and sets it so that the character is in the center of the screen. To do so, I set its *x* to the player's *x* minus half of the screen's width and its *y* to the player's *y* minus half of the screen's height.

The camera knows what character to follow because it's told in its setup event with a *player* variable. I use Clockwork's *find* function to find the character with a name I specify when putting the camera in levels.xml. By setting it in levels.xml like this I can easily choose who the camera starts with.

```
<object name="camera" type="cam" x="0" y="0"
vars='{"player":"Essie"}'></object>
```

To make it change the character it follows, I added a new event: *onAllyChange*, which, as seen before, is called every time the character in control is changed, and changes the *player* variable in the camera to the new player.

HOLDING

Throughout the game, Essie and Keystroke will need to hold each other or other things in order to combine their abilities or carry their friend or objects to places they can't reach on their own.

Figure 5: Essie and Keystroke carrying each other.

In order to hold something, a character simply changes a variable *holding* from null to the object it is holding. The real deal is the character being held. It needs to stop moving and get attached to the one holding so it has a variable *heldBy* set to that character. While in this state, it cannot change its animation from the "held" one, and, every frame, instead of moving freely, it sets its *x* and *y* to the holder's ones (with a little *y* offset to look on top) and its *z* to one behind the holder.

There are quite a few conditions on this caused by other actions but they aren't interesting enough to point out here.

HEALTH BAR

The health bar is not a very flashy feature but there is a little detail that was entertaining to program. It starts being bright green when it is full but gradually changes to red as it empties. This alerts the players that their health is dangerously low with the use of color.

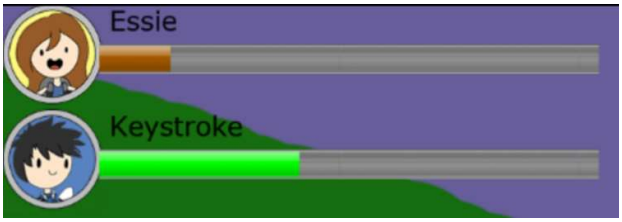


Figure 6: Health bars.

The bar starts being pure green, the color variable being *rgb(0, 255,0),* and when decreasing the health, if it is equal or lower to its 40%, I start changing the color. The *blue* value will always be 0 but *green* will start decreasing as *red* increases. To do this, I set a *green* variable to 255, which is the maximum value of each color in the RGB format, multiplied by the amount of health left, and all this divided by the 40% of the health, thus making this 40% be the maximum green value and decreasing it from there. Then the *red* variable is simply *255 – green*. This creates a smooth transition between green and red only when the health is low.

CROUCHING

Crouching is one of Essie's key abilities which she will use to get through small holes Keystroke can't go through. To crouch, a character's height is divided by half and their speed decreased. To stand up again, height and speed are restored but there is also a check that upon restoring the height the character won't get stuck in a wall. This is done with a simple *collisionQuery* to find if there are walls above.

SWIMMING

By default, all characters swim when they enter deep water. They just float on the surface of the water, being unable to dive. If they enter from the sides or bottom of the water mass, or are dragged to the bottom by another character who can dive, they go slowly upwards until they reach the surface. Apart from this, characters may behave differently when swimming like, for example, their speed can decrease with respect to running.

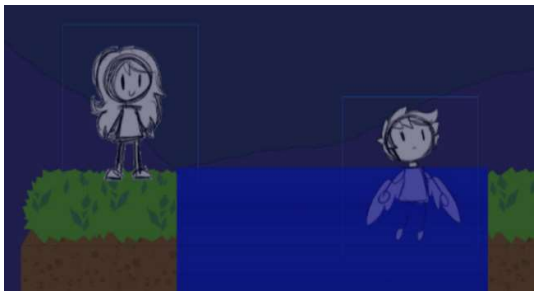


Figure 7: Keystroke swimming.

The *deepWater* component has a special type of collision, created just for it, and defined in my *customCollisions* file. This is a collision of type *waterBox* and can only collide with a point called *waterDetectionPoint*, present in all characters and set to two thirds of their height (counting from down), to be more or less around their neck. In the collision check for updating the character's position, I know the character has entered the water if there is a *waterBox* colliding with that point. If there is, the character is in the water.

Upon entering the water, the character's animation changes to the swimming one and an *onWater* event is called. This event is implemented in each character to make them have different behaviors like, for example, Essie get her speed divided by 5. The code also checks if it is the first time the character touches the water or

if it was already swimming the last frame to generate particles that mimic a splash of water. It does this too when the character leaves the water.

While being inside the water, the *waterDetectionPoint*'s *y* must be exactly the water's *y*, to make that point be in the surface of the water. For this, the character's *vy* is set to -1 until it reaches the desired position.

When leaving the water, *onLeaveWater* is called, which restores the previous conditions in the character. For instance, it multiplies Essie's speed by 5, to make it the same as before swimming.

THE MERMAID PAREO

The first of Essie's abilities obtained from a Fallen Star. The Mermaid Pareo allows her to transform into a mermaid, making her able to dive in deep water, swim a lot faster, and jump higher when doing it from water. However, she will move slower and jump less when out of the water.

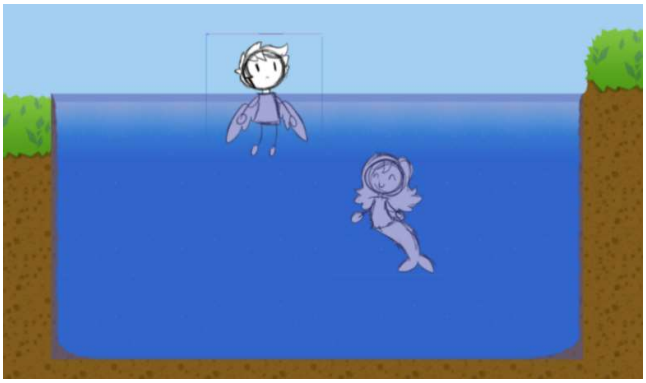


Figure 8: Essie diving.

When using the Mermaid Pareo, Essie transforms by changing its sprites to the mermaid ones, by having a special moving state that appends an "M" (of mermaid) to any move state to choose the correct one in the spritesheet. She also becomes slower and heavier. All of this is done in an event only she has. When she enters water the *onWater* event makes her a lot faster and lighter instead.

As long as she is diving, the controls change slightly as she can now move vertically without jumping. The player is now allowed to move up or down freely without being influenced by gravity until they leave the water. She can jump, however, giving her more speed upwards and allowing her to come out of the water with a strong, high spring. This moving behavior in the water is the default one for characters that can dive, and is coded in the water collision detection part of *mob*.

In this state, she can also hold Keystroke or objects and carry them deeper.

OBJECT SELECTOR

By this point I already had one of the artifacts implemented but I needed something to choose when to use it, or change between different artifacts when they are implemented. This is when I created the object selector which sits at the top right of the screen and shows the currently selected object.

Upon starting the game, a level is loaded where there is nothing but a game setup component which sets a few engine variables and loads the first real level. Between those engine variables there is one that holds the names of the objects that can be owned by both Essie and Keystroke, and another with the names of the unlocked ones. In the object selector, I loop through them when the assigned keys or buttons are pressed, checking if the character being played is Essie or Keystroke to show a set of objects or another, and if the given object has been unlocked, and sets its spritesheet state to the one showing the selected object. When

the button or key to use the object is pressed, the object selector *useObject* event is executed, which itself calls another event depending on the currently selected object.

THE DRILL HANDS AND THE INK SWORD

The second artifact implemented for Essie were the Drill Hands, bracelets that make her hands become drills she can use to attack or break cracked walls, and give her a slight speed boost in the air while the attack lasts. They cannot be used to fly a large distance however, as she needs to fall until touching the floor before she can use it again.

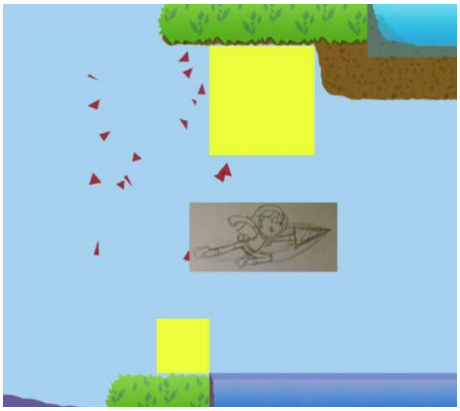


Figure 9: Essie drilling through a wall.

When the player uses the Drill Hands, control is taken away from them until the end of the attack, and Essie leaps swiftly towards the direction she's facing, breaking any cracked wall and hurting any enemy on her way. There is a timer that makes her sprite state immutable while she's executing the movement. Her *drillUse* variable is decreased by one; when it reaches zero, she can't execute any more drill attacks until it is restored when she is grounded.

The cracked walls act as normal walls but destroy themselves when they are touched by any *mob* that has a *drilling* move state at that moment. When collisions are checked in *mob*, objects with an *interactive* property are also searched. When they are, the *mob* calls the *interaction* event of the touched objects. In the case of cracked walls, the *interaction* event is the one which destroys them.

Keystroke's first Fallen Star artifact, the Ink Sword in its most basic form, holds many similarities with the Drill Hands, like its main function: he can use it to attack or break walls of plants. It allows him to execute a three-part combo attack, through which he moves slightly in the direction he is facing.

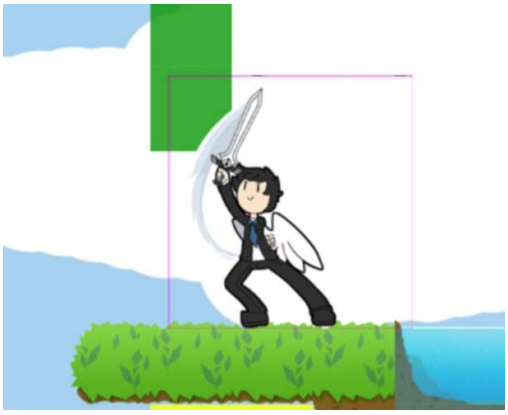


Figure 10: Keystroke wielding the Ink Sword.

Like with the Drill Hands, the player loses control over Keystroke while each part of the combo is executed, and its sprite state is also kept in the sword animation with the use of a timer. Timers also make him able to execute different attacks with the same button, changing to the next move if enough time has passed, or coming back to the first one when too much time went by since the last one executed.

Plant walls behave exactly like cracked walls do, except that their *interaction* event only breaks them if the *mob* touching them has a *cutting* move state.

PARTICLES

One of the little experiments done through the development of the project was the use of particles, which are not a needed feature for the gameplay but a flashy one, which makes the game more visually attractive. Particles were used for the water splashes mentioned before in the swimming section, in the shape of blue and white circles, as well as for when cracked walls are broken, as brown triangles, and plant walls are cut, as leaves, and when Essie transforms into a mermaid, with small white clouds and randomly colored stars with random number of sides, and when Keystroke gets petrified, with some clouds of dust.

Figure 11: Essie transforming into a mermaid with a star particle explosion.

Particles are spawned by a component which is created in the spot where the particles should come out of and with a variable *amount* which tells it how many to generate. Then each individual particle is created with a random size, speed, jumping force, direction, and rotation. Some types pick other random properties here like, for example, stars get a number of arms between 4 and 10 with greater chance of 4 and 5, and a random color chosen from an array of possible colors, which makes yellow more probable. They have a timer initialized here too which will make them destroy themselves when it's over.

At first, particles inherited *mob* but soon this component became too complex for what particles needed and so a lighter version of *updatePosition* was created in the particle component. This version does basically the same as the starting point of the *mob* one but stops checking collisions once it is grounded to make them have less impact on performance. This event is called in the particle's *#loop* event, which also decreases by one the particle's speed each frame and decreases the timer until it becomes zero and it is destroyed.

Another special kind of particle is money. When a chest is open, the money comes out of it in all directions before falling to the ground. Each of this particles is of a different kind with a certain value (decided by the developer, not random) and their weight also depends on their value, making coins and gems fall faster than bills. Apart from inheriting *particle*, these components inherit *hitbox* and are *interactive*, allowing players to pick them up in order to make them disappear and add their value to the players' money.

Figure 12: Raining money.

DAY-NIGHT CYCLE

To make the lighting in the game more interesting, there is also a component that switches between night and day. It is implemented as a timer that resets upon reaching an arbitrary number that represents the length of night and day. When reaching that number, it executes globally an event *nightTime* or *dayTime*. The *dayNightCycle* component implements these functions too. *In nightTime*, it creates three components named tint, which are blue colored rectangles that cover the whole screen (they are used for the water too as their width, height and position can be adjusted). The *dayNightCycle* controls their opacity depending on the time to make them more visible as the time passes. The *dayTime* event makes them lower their opacity until it reaches 0. But why three of them? Each of them is spawned at a different distance from the camera, making the furthest background be darker than the foreground.

Figure 13: Difference between day and night.

PETRIFIED BY SUNLIGHT

When first talking about the characters I mentioned Keystroke has a curse that makes him turn into stone when touched by sunlight. This mechanic could be easily implemented with a box collider but I decided to go a step further. Every light in the game calculates dynamically how far it reaches, having in mind all the walls that are in its way. This was done to have a more realistic lighting as well as to be able to create levels that include moving lights and require hiding behind walls or platforms to avoid them. There are even concepts of levels in which the time of the day will influence how the level must be played.

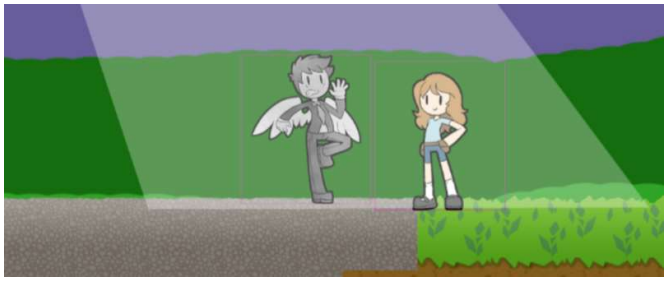


Figure 14: Keystroke petrified by the sunlight.

The algorithm used is based on the one proposed by Amit Patel for 2D Visibility in redblobgames.com[11]. This method is an additive algorithm based on raycasting. Starting at the center of the light source, a ray is casted in every angle to find the beginning and the end of walls and choosing which ones are closer to it, and then the triangles composing the visible area are filled in. As he explains, "For the area between consecutive rays, we want to find the nearest wall. This wall is lit up; all others are hidden. Our strategy will be to sweep around 360° and process all of the wall endpoints. As we go, we'll keep track of the walls that intersect the sweep line. (…) The next step is to keep track of which walls the sweep ray passes through. Only the nearest wall is visible. How do you figure out which wall is nearest? The simplest thing is to calculate the distance from the center to the wall. (…) Whenever the nearest wall ends, or if a new wall is nearer than the others, we create a triangle showing a visible region. The union of these triangles is the area that is visible from the central point."

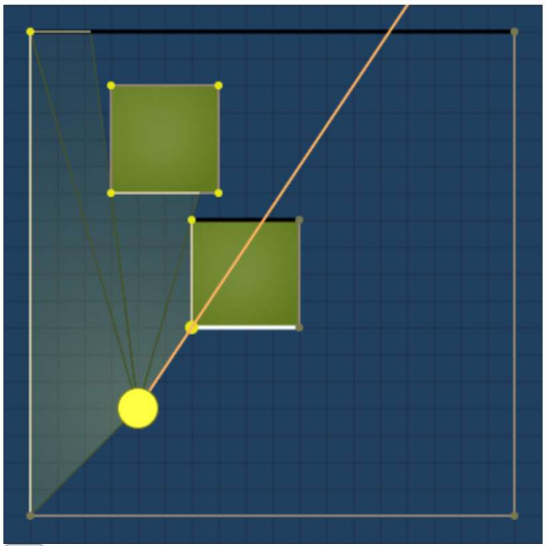


Figure 15: A ray is casted from the light source to find the nearest walls. (Image from redblobgames.com)

Having this in mind, a *light* Clockwork component was created with the following events:

- *#setup*: All the needed variables are initialized: empty arrays are created for walls and lights, the color and opacity of the light are set, etc.
- *addWall*: Adds a wall to the array of walls. The walls are JavaScript objects with x0, x1, y0, and y1 as properties, indicating the starting (*(x0, y0)*) and ending points of the wall (*(x1, y1)*). Each time I create an object which blocks the light, I call this event in its setup. I do it 4 times to add walls to each of its sides.
- *removeWall*: Removes a wall from the array.

- *addLight*: Adds a light to the array of lights. These are JavaScript objects with *x* and *y* defining their position.

- *setLightPosition*: Changes the position of a light.

- *updateLights*: This is where almost everything happens. For each of the lights the angles from their center to the start and end of each wall are calculated and then the closest points are selected and sorted from smallest to biggest angle and pushed into an array of paths that will be passed to Spritesheet.js to be painted in the canvas. The animation code of the light takes each of those paths and creates a canvas path with its points, which is then filled with the color and opacity set in the setup.

- *#loop*: Here is where the logic for petrifying Keystroke lies. Each frame I get Keystroke's position (with a little offset to take the point at the center of the sprite) and pass it to Spritesheet. The animation code for the light uses the canvas function isPointInPath to check if that point is in the path of the light. It then stores the result in another variable so it can be read in the loop. Depending on that result and the current state of Keystroke, the *petrify* event he has is called or not. This event makes Keystroke unable to move and changes his animation.

As the shadow started too far away from the edges of the walls with some textures, it felt like a good idea to expand the area the light reaches and give a nice overlapping effect on the floors.

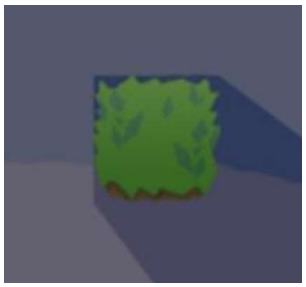


Figure 16: With some textures, the borders looked too artificial.

To do this, I took the points from the path and increased their *x* and *y* in an arbitrary quantity multiplied by the cosine and sine respectively of the angle of the point relative to the light source.

Finally, to give a better look to the light, I blurred a bit the sides to make it smoother. This was done by using the shadowBlur property of the HTML canvas context.

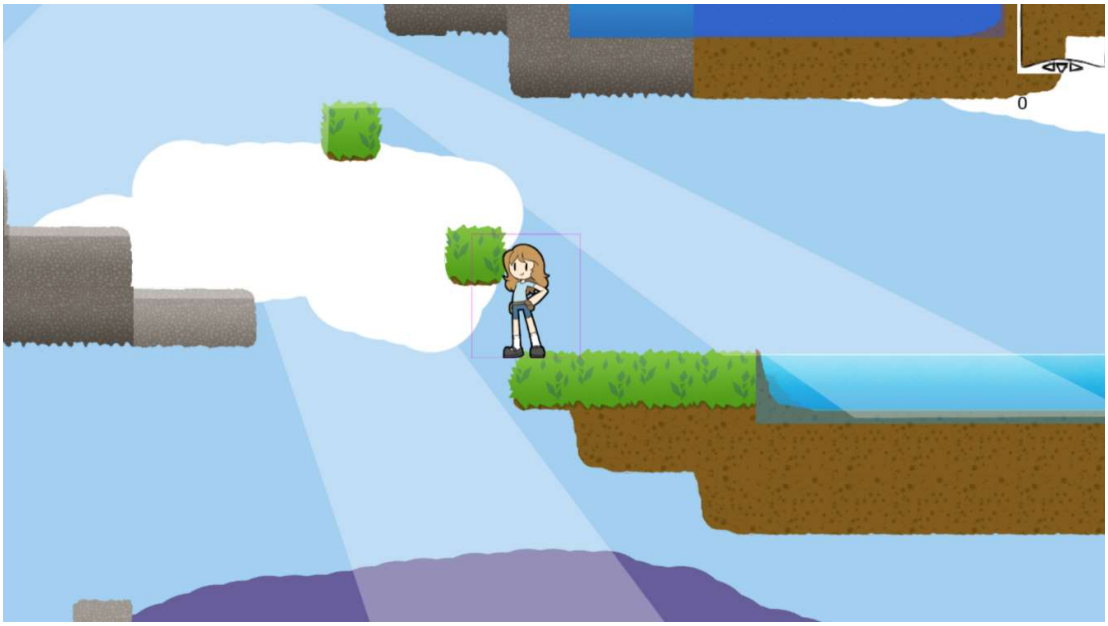


Figure 17: Final look of the light.

MAP GENERATOR

After some of these features were added, it was time to give a little of attention to the background. I wanted to have hand-made maps instead of procedurally generated ones to make them beautiful and intelligent but I did not have the time to place every floor and object in the levels.xml, counting their position in pixels. Drawing a map as an image would certainly be easier and faster than that, so that's what I did. I stopped working on the game itself to develop a tool that would make map creation a lot easier in the future. This took a few days but it makes it possible to save a lot of time now.

I created a very simple html page with JavaScript code that generates the lines needed in the levels.xml file from a given .png image in which each pixel represents a tile and the color of the pixel says the kind of tile it is. For example, the demo level available to try has this image:

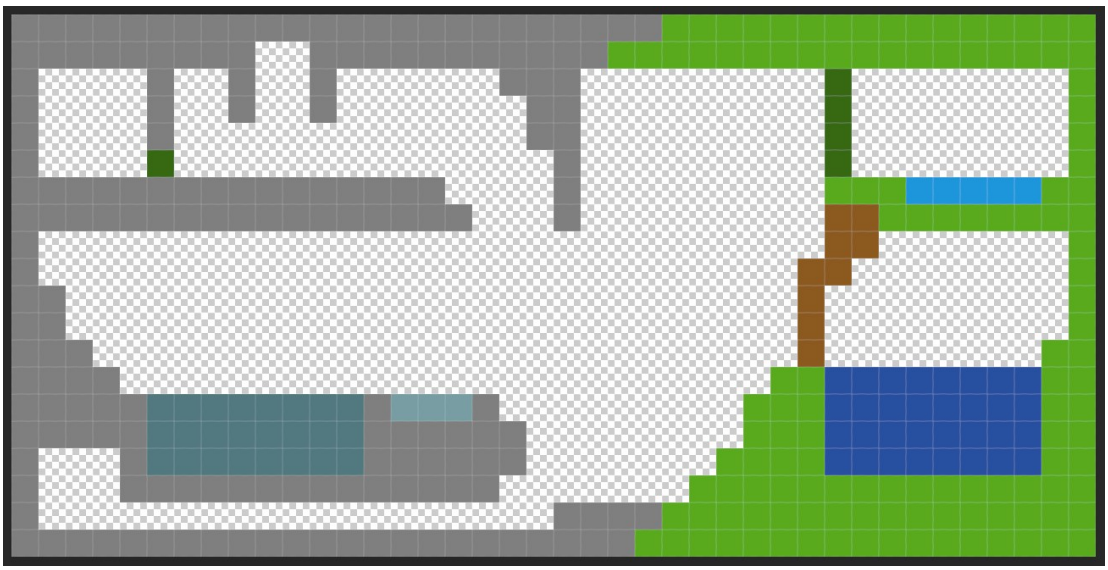


Figure 18: Demo level map.

Each of the squares is one pixel only. Grey pixels are stone, light green ones are grass, blues are different kinds of water, dark green ones are plants that can be cut, and brown ones are blocks that can be drilled.

In the .html file there is a “Get Map!” button and a text area where the XML to put in levels.xml is printed. That code needs only to be copied and pasted into the game to create the whole level.

It generates a line for each collider with its position and variables with their size and skin depending on the kind of texture I wanted. Their name is of the form X<*x* position>Y<*y* position> to make finding them manually easier.

<object name='X0Y0' type='floor' x='0' y='0' vars='{"w":2400, "h":100, "skin": "rock0"}'></object>

The page still lacks a lot of things. For now, I’ve used this tool for very few maps so the file to generate the map from, the width and height, and the size of tiles are hardcoded at the beginning of the JavaScript code but they will be able to be set in the HTML page when I start using it more.

That code starts by setting those variables. In this case, the name of the image is demoLevel1.png, my tiles are 100x100 pixels and the map is 72x48 pixels. Then I have the possible colors stored as an array of objects, similar to JSON format (JSON needs double quotes around strings so it could be easily changed to JSON by writing *“color”* instead of *color* and so on), where each entry has a color expressed as RGBA (Red, Green, Blue, Alpha), a type of object, and the skin.

```
<script>
function getMap() {
    var image = "demoLevel1.png";
    var tilesize = 100;
    var width = 72;
    var height = 48;

    /////
    var colors = [{color: "90,170,30,255", type: "solid", skin: "grass0"},
                  {color: "140,90,30,255", type: "crackedWall", skin:   "crackedWall"},
                  {color: "127,127,127,255", type: " solid ", skin: "rock0"},
                  {color: "30,150,220,255", type: "water", skin: "waterGrass"},
                  {color: "120,157,163,255", type: "water", skin: "waterRock"},
                  {color: "40,80,160,255", type: "deepWater", skin: "deepWaterGrass"},
                  {color: "82,121,128,255", type: "deepWater", skin: "deepWaterRock"},
                  {color: "55,104,18,255", type: "plantWall", skin: "plantWall"}
                    ]
    ////
```

Having that, I prepare everything to start scanning the image. I create an image with the name given before as source (I have images in the same folder as the .html), and set a few things to paint it at the beginning of the page.

Then I set a few more variables I’ll need. *last* is the last color used, *w* and *h* tell the width and height of the object I’m currently scanning, *xStart* and *yStart* are the *x* and *y* where the current object started, and *type*, *skin*, and *color* are used to indicate just what their name says.

Now I simply loop through every pixel of the image and scan with this magical function:

```
color = context.getImageData(x, y, 1, 1).data.toString();
```

getImageData(x, y, w, h) is an HTML canvas function that returns information about and image starting in the point *(x,y)* and with a width of *w* and height of *h*; in my case 1 pixel from the point I am currently on in the loop.

Knowing the color I'm working with I can know the type of tile it is. If the last color I read wasn't this color or was none, I proceed to start with an object. If the last one I was working with wasn't done, I finish it by adding the XML line to the text area (with a simple function defined separetely), specifying its name, its type and skin, where it starts, and how long it is. Then I just set the *skin*, *type*, *xStart*, *yStart*, *h* and *last* variables to the ones specified according to the given color.

The height is always 1 and only the width increases. This is because I scan the map from up to down and from left to right, being able to read only chunks in the same horizontal line. I have been working to optimize this by grouping the bigger chunks with more height but the results wouldn't improve the perfomance much without sacrifying code clarity so it is still something I will keep experimenting with in the future.

Finally, if the last color was the same as the one being read, I simply increase the width and go on.

That was the most important part of the code, there are a few more checks for special cases like finishing the object when reaching the end of the horizontal line or the end of the file but they lack interest for this document.

TILE SETTING

Once I had the map generator working the maps looked like this:

Figure 19: Blocky and boring textures.

Which is good but not good enough, they are all blocky and boring. As an artist I am, I could not accept this and I drew a collection of textures to apply to the objects depending on how they were placed.

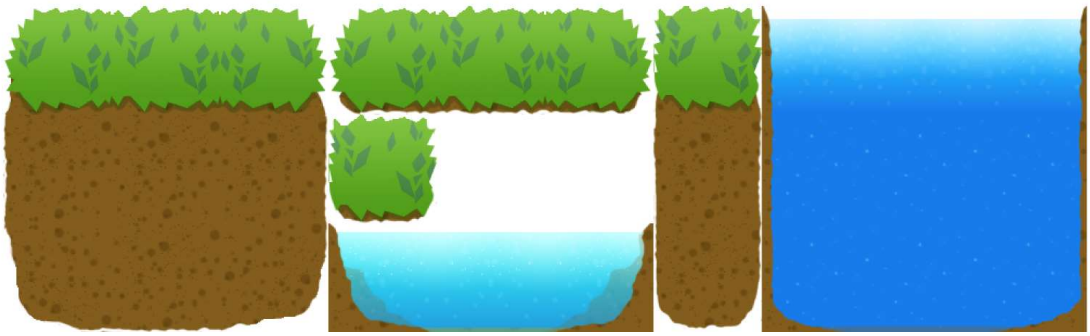


Figure 20: Different textures for different situations.

The first attempt was to decide which texture to use within the separate tool to generate the map but that lead to a week of failed attempts, destroying the clarity of the code both in the generator and the game and not achieving the effect I wanted.

The second attempt was far more successful and clean: Upon starting the level all colliders loop through their width and height creating objects which are nothing but a 100x100 image (my tile size), which have a default texture, creating this monstruosity:

Figure 21: Map before setting the right textures.

When all of these little objects have been created, another object named “bgTileStateSetter” calls globally an event all of them have that sets their sprite state to the correct one. This is done by letting all of them execute a *collisionQuery* on their four sides to check if there is something in there.

They all have a variable containing their skin set in their creation (the same skin the object had in levels.xml) and they append 4 digits to that name according to the following code where 0 means there is nothing and 1 means there is, and the order is Left-Right-Up-Down:

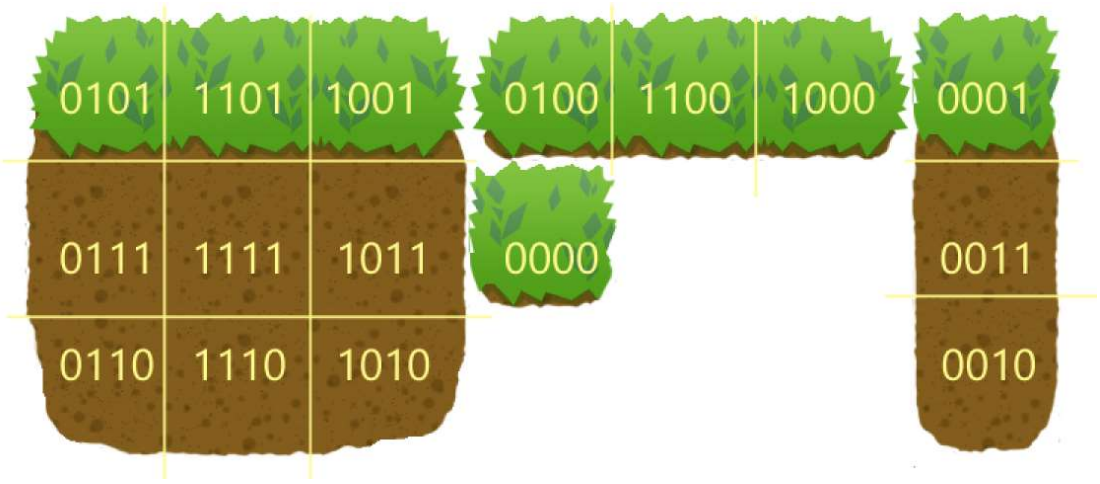


Figure 22: Collision code for tile textures.

For example, that one-tile piece of grass there would be in the state “grass0-0000”.

As easy as it sounds, this produces this beautiful results:

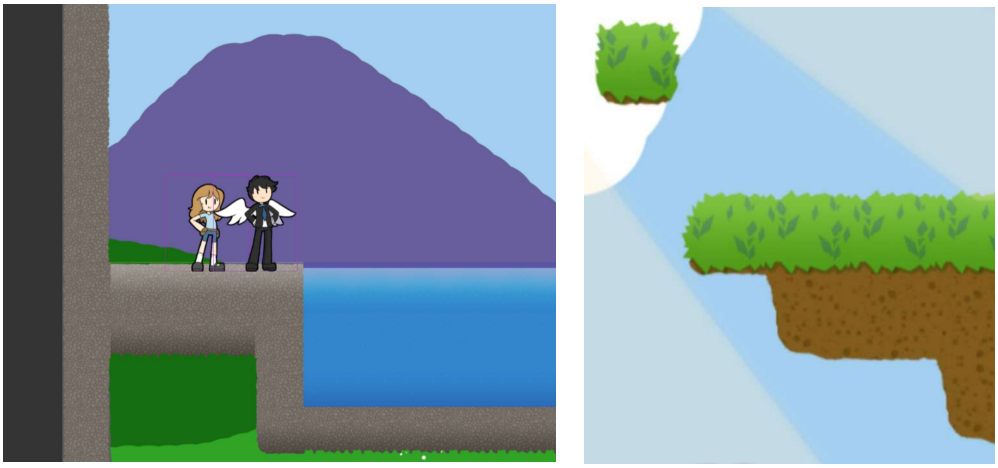


Figure 23: Textures set according to terrain.

This solution makes me able to add another nice feature: I can regenrate the textures anytime so I can make the terrain change when breaking rocks or plants with the drill and sword.

MULTIPLAYER

One of the main features Treasure Time was designed to have was multiplayer, something I waited to implement until I had clear how it would be. The approach I decided to do was to have the whole game in a first computer, which will be the server, and it will send all the rendering data to a second computer, the client, which will mirror the server’s screen but with the camera focused on the second character. The second player will play on their computer, and the keys or buttons they press will be sent to the server, to move the character accordingly.

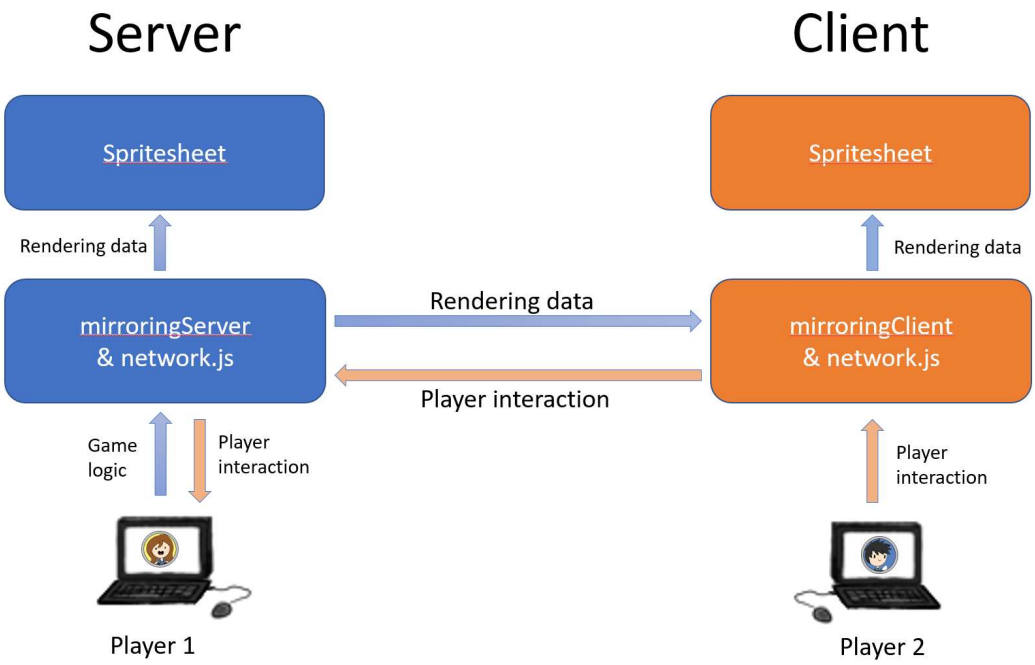


Figure 24: Mirroring diagram.

The first step for this implementation was to create a pair of rendering libraries which go before Spritesheet and process the rendering data before sending it to Spritesheet. The *mirroringServer* simply takes the rendering data and passes it to Spritesheet but also takes care of registering the function to send this data to the client. The *mirroringClient* takes the rendering data and sends it to Spritesheet. However, the setCamera function of the mirroringServer is not sent exactly how it is; instead, the camera position for the second player is passed each time the second character moves and the camera changes for the first character are ignored.

Having these, I created a *network* file with two components: *server* and *client*. Then a TCP socket was created in the server, listening on port 8776 (a random port number that wasn't being used for anything else), which, upon being accepted, creates output and input TCP sockets, and starts sending messages. To do this, it creates a function sendMessage, which stringifies the message to send the string using the output socket, and registers this function in the rendering library. The component also counts with an array of cached messages, where all messages before the connection is established are stored, and they are sent at this point. After this, it starts reading to receive messages from the client that contain the player input and calls the events to move the second character.

The client creates its own TCP sockets to connect to the same port and registers a send function in the same way. It starts reading with a very similar function and calls the appropriate rendering library functions according to the content of the message. This way, every command is received by the client and the rendering library paints it. Apart from this, the client listens for player input and sends whatever it receives to the server for it to process it.

For the client to know the server IP to connect to, I coded them to create a UDP socket and join a multicast group so that the server can pass the IP to the client through it.

Currently, server and client can be set in different computers, making everything done in the server appear in the client and processing every movement of the client in the server without much lag.

INTEGRATION, TESTS AND RESULTS

During the development of the game, each new feature has been tested through its process and in combination with the rest of implemented features once it was done. Special levels were made for each of the features to be tested easily and in different scenarios.

At the beginning of the project the level was nothing but a few solid platforms at different heights to test the character moving and jumping. A few more platforms were added way higher later, when working on Keystroke, and the level stayed like this for testing the character changing, to have a long distance between Essie and Keystroke and check the camera worked correctly, and for holding, to have Keystroke picking Essie up and flying her to the highest platforms. This level stayed too for testing the health bar and the fall damage, making Key fly Essie at different heights and letting her go until she hit the ground. Ouch. This stage of development became a really fun one where I played to grab Essie and let go of her to fly down faster and save her from harm. It was at this point that I knew the game would really be entertaining, as a simple feature made me have a lot of fun.

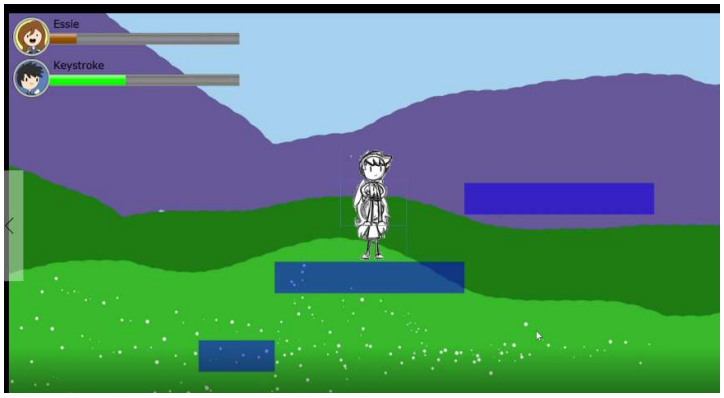


Figure 25: Essie holding Keystroke in the first testing level.

After these, the height of one of the platforms was lowered to make it have just one tile of free space between it and the ground, making it a perfect place to test crouching. It first was tested in what would be a normal use case, crouching, moving through the space, and then standing up. Having that working, I started crouching mid-jump, falling, crouching in a platform and walking towards the border to fall, standing up below the platform to get stuck, … and fixed all the undesired behaviors some of those situations caused.

When I implemented swimming is when most of the problems appeared. I added to that level a big square of water and tested all that had been implemented before in it. It turns out water is a big bug source and so, most actions from this point of development on cannot be executed in the water. This isn't a big loss though, as most of them don't make much sense in the water anyway. Crouching is one of them.

The next step was the Mermaid Pareo, with which I used the same block of water. It was also a hard feature to implement because of the special conditions of water but, as most actions could not be performed while in the mermaid form, the integration wasn't as hard as with swimming.

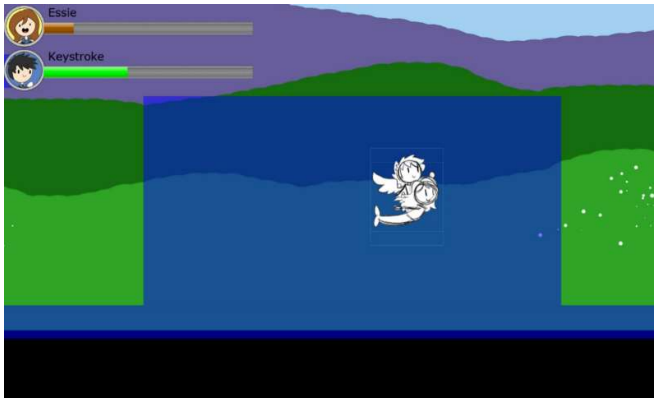


Figure 26: Holding working perfectly while diving in the second testing level.

I kept adding more things to this level as I developed. After the water, I placed some cracked walls to the right to test the Drill Hands, which resulted in me ending up in the water while using them and discovering more bugs that needed fixing. After this, I made these cracked walls explode in money when broken to test the particles, which resulted in too many objects checking collisions at once, affecting performance greatly. That is when I made them have a simple version of *mob* and stop checking collisions. Later, I added some plant walls right next to and above the cracked walls to test the sword.

For implementing the map generation and tile setting I created a few levels from scratch that would show all the cases in which the tile textures could be and with all the types of tiles I had drawn at the time. It was very useful to confirm the cases I had planned worked and to find cases I had not thought of while seeing them in action in the game, showing what would be the final look of a level. This level ended being a good place to test the lighting too so I did it here, seeing how the light behaved with the different platforms.

Figure 27: Third testing level with one of the first versions of light.

Keystroke's petrifying, however, had been tested before in the previous map which unveiled another bug associated with water. Something made petrified Keystroke splash infinitely. It also made me think about what should happen with petrified Keystroke in the water. Should he stay in the surface, drown as a stone should do but destroy some mechanics, or perhaps turn back to normal? In the end, I decided to keep him in the surface.

After all of these, I created a demo level to use whenever I need to show the game (the one you can find in the notes section of the introduction of this document) and that is the current level I am now using for all the new features and for testing the multiplayer.

With respect to multiplayer, the original approach consisted on creating a UDP socket in both server and client and make both of them join a multicast group through which the server would send its IP to the client. After it was all coded, I tested it with two computers and failed to find the reason why it wasn't working. I finally discovered that my router decreases the TTL (Time To Live) by one, even in the same network. As I am unable to set this field to more than 1 in my tests, this kills the messages before they reach the client. I continued working with the hardcoded IP and I leave this issue to be fixed in the future.

CONCLUSIONS AND FUTURE WORK

CONCLUSIONS

After working on this game for so long, I have learnt that the real difficulty of game development is in the details. The basic gameplay is usually easy but the details like the particle effects, staying in the surface when swimming, changing the sprite states to best fit the movements, making the images of the tiles depend on their surroundings, etc… is where most of the work is. The basic algorithm for most of the things tends to be easy to explain but making it work with the rest of the features requires many little conditions that have to be checked for it to work properly.

Performance also becomes an important problem as more features are added. Some implementations need to be changed or even redesigned to make the game faster, requiring various iterations. Being made with an engine in development, any performance problems were informed to the creator, who could many times improve the engine to make it more efficient for all users.

Talking about the engine, in the process of the game development, some bugs were found that sometimes difficulted my work and I also had to migrate the game from the first version of the engine to the current one mid-development. However, these conditions weren't much of a hassle and instead contributed to making Clockwork a lot better.

I have also seen how important it is to plan everything before starting to program to make the code cleaner and more efficient and not get distracted with imaginary scaling problems. In the first prototype of the game everything was being made very abstract to make it able to be applied to any character. This soon made the code very messy and it wasn't that much of a need. Instead, it was redesigned to be more focused on the characters I knew I'd have.

All in all, I learnt a lot in the development of this game; it challenged and entertained me in many ways but I am very proud of the result and looking forward to finishing it.

FUTURE WORK

Although the game already has a lot of working features, there is still a lot of work for this game to be finished the way I want it to be:

- Many artifacts' abilities still need to be implemented for the game to be more complete.
- A lot of levels have to be designed although, once designed, they'll be easy to put in the game with the tools I have already developed. Many enemies and objects need to be designed or made art for as well.
- Many things like interface and some character movements still need art.
- There will be a city in which there will shops to buy healing objects and the like. Its implementation has already been started.
- The multiplayer part still needs to be perfected and the problem of getting the IP needs to be fixed.

REFERENCES

[1] Official Clockwork.js website http://clockwork.js.org/

[2] Clockwork documentation on spritesheets http://clockwork.js.org/documentation.html#spritesheets

[3] Imagine Cup official website https://compete.imagine.microsoft.com/en-US/imaginecup?wt.mc_id=DX_874656

[4] Game Development World Championship website https://thegdwc.com/

[5] Intel Level Up Contest https://software.intel.com/sites/campaigns/levelup2017/?utm_campaign=spredfast_SSG-Tech-IoT*SSG-Content-DevExperience&cid=sm_twitter_spredfast_SSG-Tech-IoT*SSG-Content-DevExperience_sf61523215&utm_source=twitter&utm_content=sf61523215&utm_medium=social&sf61523215=1

[6] Steam metroidvania search http://store.steampowered.com/tag/en/Metroidvania/#p=0&tab=TopSellers

[7] Shantae: Half-Genie Hero Steam stats http://steamspy.com/app/253840

[8] Owlboy Steam stats http://steamspy.com/app/115800

[9] Ori and the Blind Forest Steam stats http://steamspy.com/app/261570

[10] Xbox Play Anywhere http://www.xbox.com/en-us/games/xbox-play-anywhere

[11] 2D Visibility Algorithm by Amit Patel http://www.redblobgames.com/articles/visibility/

GLOSSARY

UWP	Universal Windows Platform, application architecture designed to allow applications to run on any Windows 10 device.
JSON	JavaScript Object Notation, a human-readable and machine-readable way of formatting data.
XML	eXtensible Markup Language, a human-readable and machine-readable way of formatting data.
API	Application Programming Interface.
Game Engine	Framework for game development.
Platformer	A game genre characterized by its side view and having a gameplay based on jumping through platforms.
Metroidvania	A subgenre of action-adventure games with features reminiscent to the games *Metroid* and *Castlevania*. Includes big maps players can roam through freely and unlockable abilities.
Sprite	A 2D game image.
Spritesheet	An image with several sprites to be used in a game. Usually, of the same character in different states.
Mob	A term for mobile characters in a game.
Canvas	The HTML5 canvas, where JavaScript can be used to draw.
TCP	Transmission Control Protocol, provides reliable data delivery between machines connected by an IP network.
UDP	User Datagram Protocol, provides data delivery without the overhead of TCP but may lose packages.
Lag	Delay between the actions performed in the server and the results in the client.
Git	A version control system.
Kickstarter	A website for crowdfunding.
Indie	Independent. Games created without financial help of large companies.
Steam	A software distribution platform by Valve.
Humble Store	A software digital store.
Indiebox	A software digital store for indie games.

Good Old Games	A software digital store.
Xbox One	The latest generation console by Microsoft.
PlayStation 4	The latest generation console by Sony.
PlayStation Vita	The latest generation portable console by Sony.
Nintendo Switch	The latest generation console by Nintendo.
Nintendo WiiU	The predecessor of the Switch, created by Nintendo.
HoloLens	Mixed reality headset by Microsoft. Shows holograms in the real world.
Traditional animation	Animation where all frames are done by hand.

ANNEXES

A. GAME CONTROLS

The game can be controlled with both keyboard and gamepad. These are the controls:

Key	Action
Left/Right arrow keys	Move left or right
Up arrow key/Space	Jump
Down arrow key	Crouch (Essie only)
W	Change character
E	Hold/Release character
A	Previous object
S	Use object
D	Next object